I-CoPES: Fast Instruction Code Placement for Embedded Systems to Improve Performance and Energy Efficiency

Sri Parameswaran
Dept. of Computer Science & Engineering
The University of New South Wales
Kensington, NSW 2052
e-mail: sridevan@cse.unsw.edu.au

Jörg Henkel
C&C Research Labs
NEC USA Inc.
Princeton, NJ 08540
e-mail: henkel@nec-lab.com

Abstract

The ratio of cache hits to cache misses in a computer system is, to a large extent, responsible for its characteristics such as energy consumption and performance. In recent years energy efficiency has become one of the dominating design constraints, due to the rapid growth in market share for mobile computing/communication/internet devices.

In this paper we present a novel fast constructive technique that relocates the instruction code in such a manner into the main memory that the cache is utilized more efficiently. The technique is applied as a pre–processing step, i.e. before the code is executed. It is applicable for embedded systems where the number and characteristics of tasks running on the system is know a priori. The technique does not impose any computational overhead to the system. As a result of applying our technique to a variety of real-world applications we measured (through simulation) that the number of cache misses drops significantly. Further, this reduces the energy consumption of a whole system (CPU, caches, buses, main memory) by up to 65% at an only slightly increased memory size of 13% on average.

1 Introduction

Performance can be enhanced with the addition of some overhead by the reduction of cache misses. In embedded systems reducing cache misses improves performance allowing the system to meet deadlines at reduced cost. Reduction in cost is attributed to a lower cost processor. Reducing cache misses reduces the total amount of energy consumed by the system. When the number of the cache misses is reduced, the bus from the cache to memory is exercised less frequently, which in turn reduces total system energy consumption. For off chip memories it has been estimated that the total power savings can be up to 70% of the total power [2].

Cache misses can be classified as follows: compulsory misses, conflict misses, and capacity misses. A compulsory miss occurs when an instruction or data is referenced for the first time, a conflict miss when data or instruction compete for the same memory location, and a capacity miss when the size of the cache is too small for the amount of memory needed to execute the program under consideration.

New processors demand efficient cache architectures. To satisfy the fast access demands of the system, direct mapped caches are commonly used. While these satisfy the speed constraints, they increase the amount of conflict misses, since there is no alternate position for the requested data or instruction to occupy in cache (as is the case in set associative cache design).

Modern processors contain both an instruction and a data cache. There have been several differing methods to reduce cache misses in both.

In the I-CoPES methodology, which is described in this paper, a scheme is described to reduce the instruction miss conflict misses. The method uses a constructive algorithm to reorder the mapping of basic blocks in the cache, so that the total number of misses are reduced. Another heuristic algorithm is used to map those basic blocks to the main memory from the allocated cache locations reducing the number of conflict misses. This paper further presents the energy savings, which can be obtained by reducing conflict misses. The system model here assumes that all of the components of the system are on chip, and therefore, for the first time we will be showing the total energy savings that can be achieved by reducing cache misses through instruction code placement for systems on a chip. For the examples given in the paper, the cache misses can be reduced on average by 32%, and the energy consumed can be reduced by up to 65%, On average, the required increase in memory was about 13% and varied from 3-38%.

1.1 Related work

Several articles have appeared in recent literature about reducing cache misses by reorganizing data or instructions in the cache. The work on cache misses has predominantly concentrated on the data cache optimisations [5]- [11][27][26]. The instruction cache optimisation methods have been around for more than a decade and are described below.

Hwu et al, in [1], effectively reduced the cache miss rates in a compiler called IMPACT- 1 by function in-line expansion, trace selection, function layout and global layout. Function in-line expansion replaces the function calls with the functions in higher execution calls. Trace selection groups the basic blocks which are often executed together, which then reduces the compulsory and conflict misses. Function layout places the most important descendant after the function entrance, thus reducing the conflict misses. Global layout places functions, which are executed together in close proximity to each other. McFarling in [3] and [4] , analysed functions such that the dependencies amongst functions were exposed and exploited in order to reduce misses. Chow [20] reduced cache conflicts by sorting functions by their execution frequencies, and then grouping functions together to reduce conflict misses. All of the above methods worked at the functions level.

The first of the global methods was proposed in [25] by Tomiyama and Yasuura , where an ILP formulation was applied to two differing methods in order to reduce the cache misses. The first method applied trace selection, which reduced the cache size but increased memory size. The second method, was a refined method and applied trace selection, trace merging and trace placement in order to reduce the total misses. The ILP formulation reduced the speed of application and is not a suitable method for large program optimizations. Performance estimation of such caches has also been reported in the literature [24]. Kirovski et al [23] use the frequency of execution to reduce cache misses in a system to be synthesized. In [21] and [22] the authors used a scheme where half the cache

0-7803-7247-6/01/$10.00 © 2001 IEEE

was assigned for high priority tasks and the other half was allocated for non-high priority tasks. This method was applied to instruction cache optimisation in multi–processor systems by Li and Wolf in [12], though they later abandoned it in [13] for random placement of instructions in memory and doubling of cache sizes until deadlines were met. In [29], a constructive method is given to place tasks in memory for multi–processor systems.
Other approaches that are relevant to our work stem from the area of system-level power estimation and optimization: Dave et al. [14] introduce a task-level co-design methodology that optimizes for power consumption and performance. The influence of caches is not taken into consideration. The procedure for task allocation is based on estimations for an *average* power consumption of a processing element. The approach described by Hong et al. [15] uses a multiple-voltage power supply to minimize system-power consumption. Another system-level power estimation approach that focuses on peripheral cores within SOCs is described by Givargis et al. [16]. They presented a hybrid approach that uses one-time-obtained gate-level power data and propagates it to an executable specification in order to speed up power estimation. Simunic et al. [17] simulated the power consumption of an ARM processor plus a cache hierarchy and a main memory using a cycle–accurate approach. Lajolo et al. [18] have conducted research on a cycle-based co-simulation environment for power estimation.

1.2 Relevance of this work

In the work described in this paper we give a constructive algorithm, which works effectively, to reduce conflict misses. The methodology increases the total amount of main memory. A second algorithm is given to reduce the total amount of used main memory. For the first time in the instruction cache optimisation literature, we give a methodology to show the effect of code placement in both energy consumption and performance in whole systems.

1.3 Assumptions

1. The size of the task is no bigger than the size of the cache. This assumption is quite valid in embedded systems where the tasks are usually small enough to fit into small cache sizes. If the task is too large for the cache it is possible to break up the task into smaller granules such that each granule will fit into the cache.

2. Only Level-1 caches are available for use. Once again in an embedded system, where frequently there is no cache at all, it is unlikely that more than a single level of cache is going to be available for use.

3. The caches are direct mapped. High-speed systems frequently use direct mapped systems in order to speed up the system as much as possible. This assumption makes it easier to analyse due to the deterministic mapping to cache from memory.

4. The instructions of a task are allocated to a contiguous region of memory. This states that a task mapped to cache will be in a continuous region in that cache (except when it overflows the cache, which then will continue from the top of the cache). If the memory is going to be in two parts then they will be considered as two separate tasks.

5. All addressing performed within each task uses relative addressing (i.e. branch) only. Tasks can be mapped to arbitrary locations within the memory map. This does not allow absolute addressing. If absolute addressing is used then instruction code cannot be loaded into any section of the code. If absolute addressing is necessary then we need to have a more sophisticated compiler.

6. The problem is sufficiently large so that the total size of the instructions are several times larger the size of cache. This is a reasonable assumption in a realistic system.

Table 1: A Task mapping Example

BB	Freq	Mem Map	I-Ca Map	New I- C Map	New Mem Map
1	25	0-149	0-149	1-149	768-917
2	300	150-259	150-3	146-255	402-511
3	500	260-299	4-43	0-39	0-39
4	25	300-474	44-218	81-255	1105-1279
5	25	475-549	219-37	181-255	1461-1535
6	10	550-674	38-162	131-255	1667-1791
7	10	675-779	163-11	151-255	1943-2047
8	500	780-919	12-151	40-179	40-179
9	300	920-1049	152-25	146-255	658-767
10	500	1050-1120	26-96	180-250	180-250

1.4 Motivational Example

In systems where cache is present, the instruction is first brought to the cache before the instruction is executed. In an embedded system, often a single application is repeatedly executed with separate data sets. In general purpose computer systems blocks are compiled and stored into memory in a first in first out (FIFO) scheme, which means that when a block is mapped into the cache, we have little control as to where the block will end up in the cache. However, by placing blocks carefully in memory, we could make critical blocks at least partially stay in the cache at all time. So that the instructions (after the first time when compulsory misses occur) will most probably be in the cache at all times.
In the example in Table 1, there are 10 basic blocks and the frequency of execution of each block is given in column 2. In column 3, the memory mapping for the First in First Out (FIFO) scheme is given, with the corresponding cache mapping (cache size of 256) in column 4. Let us assume that three loops have been identified. Loop1 containing basic blocks 3, 8 and 10 which is executed 200 times; loop2 containing basic blocks 3, 8, 10, 2 and 9 which is executed 300 times; loop 3 containing blocks 1, 4 and 5 which is executed 15 times; and finally loop 4 containing blocks 1, 4, 5, 6 and7 which is executed 10 times.
If loop 1 was considered, block 3 is between locations 4 and 43 in cache, block 8 is between locations 12 and 151 and finally block 10 is between locations 26 and 96. Thus every time a block is brought into the cache another block (at least partially) has to be ejected from the cache. If on the other hand, the mapping given in column 5 is used, it can be seen that block 3, 8 and 10 reside in separate parts of the cache and therefore there will be no conflict misses. Taking loop 2 into consideration, it can be seen that blocks 3, 8 and 10 are the predominant blocks and they are laid out in the cache as before (since they can only occupy one set of locations). Since the sizes of blocks 2 and 9 are rather large, it is clear that wherever we allocate them in cache there is going to be some misses in the cache. Therefore we move both of these blocks to the bottom of the cache so that at least some part of the more predominant tasks (namely 3 and 8) are not replaced. This type of strategy is justified, since in most of the code we observed, the loops such as loop 1 and loop 2 were usually close together and were executed one after the other.
This strategy of constructively relocating code and then mapping it to main memory (see column 6), is an example of the method

used in this paper.

2 Problem Statement

Let the size of the cache associated with processor P be equal to S. The basic blocks to be placed in the cache be $b_1, b_2......b_n$ all of which belong to the set B. Each block b_i is associated with size s_i. Let a loop l_r be defined as being a subset of B and the loops be numbered from $l_1, l_2, ...l_r, ...l_m$. Note that while multiple loops can share a basic block, the set of basic blocks within a loop will be unique to that loop. Some basic blocks will not belong to any of the loops. The problem is: to arrange the basic blocks of each of the loops (and other basic blocks which do not belong to any of the loops) in main memory so that it can be brought into the cache such that the conflict misses are reduced; and the total main memory used is minimized.

Complexity of the problem

It can be shown that a special case of the problem, where a basic block b_i can only belong to a single loop l_k, can be shown to be a bin-packing problem. Therefore this problem is at least as hard as the bin-packing problem which is NP-Hard. To solve this problem in reasonable run-time it is necessary to use some type of heuristics.

3 Allocation of basic blocks in Cache and Memory

This methodology contains two algorithms. The first algorithm places basic blocks in the cache so that basic blocks with high frequency are swapped out as little as possible. The second algorithm takes the placed basic blocks and maps them into main memory. Both of these algorithms are performed as a preprocessing step, taking the application's original instructions in memory and remapping them to different locations.

In order to remap instructions, it was necessary to identify basic blocks. We identified them by running the application through a instruction set simulator and finding blocks of instructions which were always executed together. The number of basic blocks within applications under consideration varied from 100 - 900.

3.1 Cache allocation

The methodology used for ordering basic blocks in the cache is as follows (see Figure 1). All the loops containing a particular basic block are grouped into a single super loop. Thus a loop will be only a member of one super loop. Each super loop's frequency (f_{sl}) is defined as the addition of all the frequencies of the component loops. The super loops are ordered in descending order of frequency. The ordered super loop list is given as $sl_1, sl_2, ...sl_p$.

For super loop sl_1, the basic blocks within it are taken in order (from highest to lowest frequency of execution of basic blocks - f_b) and these basic blocks (only whole basic blocks are allowed) are allocated to the cache from the lowest address to the highest until the cache is completely filled or there are no remaining basic blocks within that super loop. Once this step is finished, and if there are any remaining basic blocks, we find the largest basic block from the remaining basic blocks of sl_1. This large basic block is allocated to the bottom of the cache say with starting address A_{ls} and ending at the end of the cache (step 2). After this we take the next largest basic block and allocate its starting address in the cache to A_{ls} (step3). The ending address will be less than the final address of the cache. Thus if another unallocated basic block can be found which can go into the space (below the basic block we just allocated, and above the last cache address), we allocate that basic block into the available space (step 3 contd). We keep doing this until we reach the end of the cache. We take the next largest unallocated basic block (step 4), and start it at address A_{ls}

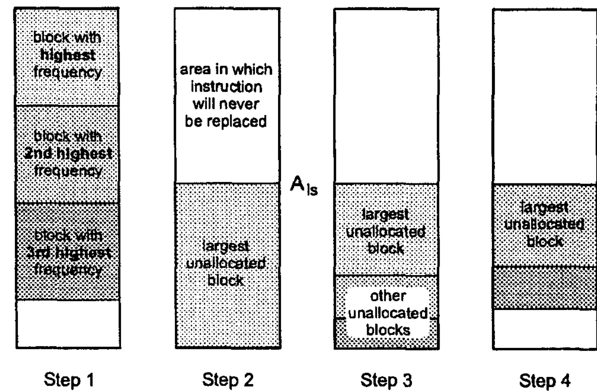

Figure 1: The cache allocation method

and we repeat the process until all basic blocks are allocated. This is then repeated to the other super loops in the ordered list.

3.1.1 Algorithm

Algorithm for ordering tasks in cache

For Each *super loop in ordered list* {

Until *Cache is filled* {

Allocate basic blocks to the cache in descending order of frequency f_b until no more blocks can be allocated

}

Reorder *Unallocated basic blocks in order of size and place in list BB_u ($BB_u = bb_{u1}, bb_{u2}, bb_{u3}...bb_{uy}$, where y is the number of unallocated tasks for that super loop)*

Allocate bb_{u1} *from address A_{ls} to end of cache (where $A_{ls} = S$ - sizeof(bb_{u1}))*

Remove bb_{u1} *from* BB_u

Repeat *until all tasks are allocated* {

Find *the next largest unallocated Task bb_{up} from the list BB_u*

Allocate bb_{up} *from address A_{ls} to A_{le} where $A_{le} = A_{ls} + sizeof(bb_{up})$*

Mark bb_{up} *from list BB_u as allocated*

Move along the list BB_u and place as many tasks as possible between A_{le} and S

Mark placed tasks as allocated

}

}

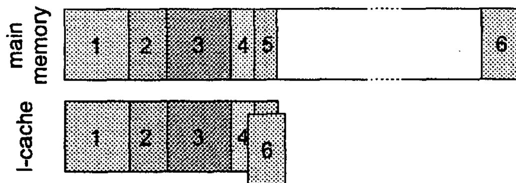

Figure 2: The memory allocation example

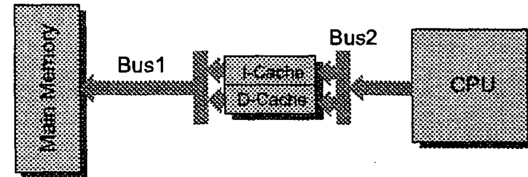

Figure 3: Architecture our method is applied to

3.2 Memory Allocation

The memory allocation algorithm takes the already placed basic blocks in the cache and directly maps them to the memory. Figure 2 shows an example of how the basic blocks are mapped to memory from the cache. In this figure blocks 1 to 5 are mapped directly on to the memory, but the block 6 is mapped to some memory locations further away, such that the mapped block will end up in the desired position in the cache. Thus if a basic block is mapped to the location from t_x to e_x in cache of the processor, then the basic block can be placed in memory in any one of the address ranges from addresses $t_x + i * S$ to $e_x + i * S$, where i is a positive integer. However, since tasks in the cache will wrap around the cache, an offset Z_r, can be added to each basic block allocated from super loop sl_r, and the basic block can be placed from memory location $t_x + Z_r + i * S$ to memory location $e_x + Z_r + i * S$. This introduction of the offset allows the reduction in size of the total memory needed for the system.

The algorithm works as follows. The super loops are ordered in descending order of total basic block size and put in a ordered super loop list. The super loop with the highest sum of all basic block sizes will be the first in the ordered list. The super loop with the lowest sum will be the last element in the ordered list. The basic blocks allocated in the first super loop are mapped to the main memory as follows: the first basic block allocated in cache is directly mapped to the memory (in this case, basic block $t_x = 0$ and $e_x = sizeof(basicblock1) - 1$); basic block 2 and upwards will be mapped by moving i along the non-negative integer range and trying to find an area in memory which is free from locations $t_x + i * S$ to $e_x + i * S$. The basic blocks allocated to the subsequent super loops are allocated thus: the largest basic block allocated to the super loop is put in the first available memory block where the basic block will fit (say M_x to M_y); from this the offset Z_r is calculated, and $Z_r = (M_x modS) - t_x$. Using this Z_r, all of the other basic blocks allocated to the super loop are mapped to the memory in the order of size from largest to smallest using the mapping each basic block from memory location $t_x + Z_r + i * S$ to memory location $e_x + Z_r + i * S$.

3.2.1 Algorithm

Let us assume that the super loops used in the system are $sl_1, sl_2, sl_3 ... sl_k$. The associated cache is C. For each super loop sl_r, the basic blocks to be executed in that super loop are as follows: $bb_1, bb_2, bb_3 ...... bb_m$ where m is the number of basic blocks.

> **Reorder** *super loops from largest sum to smallest sum of total basic block size and name them* $sl_a, sl_b, sl_c ...$
>
> **For** *all basic blocks ordered in the cache allocation order in* sl_a *do* {
>
> $i = 0$
>
> **While** *basic block is not allocated* **do** {
>
> **If** *memory locations* $t_x + i * sizeof(cache)$ *to* $e_x + i * S$ *is free* **then**
>
> **Map** *basic block to address* $t_x + i * S$ *to* $e_x + i * S$
>
> **Else**
>
> $i++$
>
> }
>
> }

For *all basic blocks ordered in descending order of size in the next super loop until the end of the super loop list* **do** {

> **Allocate** *largest basic block in the first available contiguous memory block* (M_x *to* M_y)*, which will hold the basic block*
>
> **Calculate** $Z_r = (M_x modS) - t_x$*, where* t_x *is the address in which the basic block being allocated starts in the cache at address* 0 *and* sl_r *is the present super loop under consideration*
>
> **While** *basic block not allocated* **do** {
>
> **If** *memory locations* $t_x + Z_r + i * S$ *to* $e_x + Z_r + i * S$ *is free* **then**
>
> **Map** *basic block to address* $t_x + Z_r + i * S$ *to* $e_x + Z_r + i * S$
>
> **Else**
>
> $i++$
>
> }

}

4 Validation

In order to verify the usefulness of the I-CoPES approach we had to provide a simulation environment that captures the behavior (timing true and data true) of a whole sub-system comprising a CPU, the caches, the main memory and the buses in between (see Fig. 3). The simulation environment, shown in Fig. 4, takes as an input the application program plus typical input stimuli data (as far as the application characteristics, i.e. run-time, instruction cache access, depends on the input stimuli). A trace simulator (QPT, see [19]) generates the instruction traces and feeds them into a cache simulator (Dinero III, see [19]). The output of the cache simulator are cache miss/hit ratios that are fed into analytical power models for estimating the respective energy data. for the I-cache, d-cache, buses and main memory whereas the power

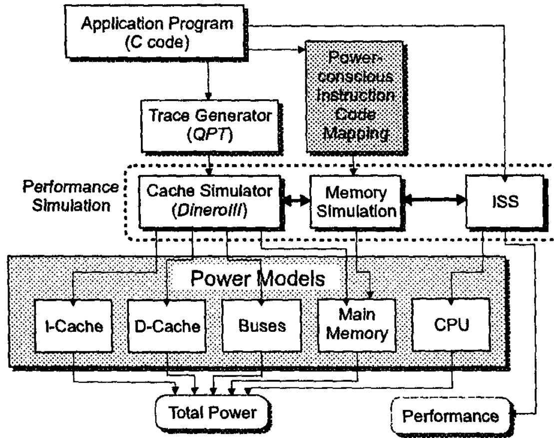

Figure 4: Experimental setup that allows to simulate the power/energy consumption and performance of the whole system

estimation of the CPU is accomplished by a an instruction set simulator (ISS) that is coupled to a table that contains energy data for each instruction and addressing mode. The following equation summarizes the way in which energy consumption of the application running on the CPU is obtained.

$$
\begin{aligned}
E_{prg} &= T_{w_c} \cdot V_{DD} \cdot \sum_{i=0}^{N-1} (I_{instr,i} \cdot N_{cyc,i}) + \\
& T_{cyc} \cdot V_{DD} \cdot (\underbrace{N_{miss,rd} \cdot N_{cyc,rd_pen} \cdot I_{instr,nop}}_{\text{data write miss penalty}} + \\
& \underbrace{N_{miss,wr} \cdot N_{cyc,wr_pen} \cdot I_{instr,nop}}_{\text{data write miss penalty}} + \\
& \underbrace{N_{miss,fetch} \cdot N_{cyc,fet_pen} \cdot I_{instr,nop}}_{\text{instruction fetch miss penalty}}) \qquad (1)
\end{aligned}
$$

where V_{DD} is the voltage supply, I_{instr} is the current that is drawn during the execution of instruction i at the processor pins, $N_{cyc,i}$ is the number of cycles the instruction i needs for execution and N is the total number of instructions of the program. T_{w_c} is the execution time of the application (see below).

The three additional shares within the brackets refer to the energy that is consumed during the penalty cycles when there is a data cache write miss, a data read miss and an instruction fetch miss, respectively. For more info on these models please refer to [28]. The total execution time T_{w_c} as it appears in Eq. 1 is obtained as follows:

$$
\begin{aligned}
T_{w_c} &= T_{w/o_c} + T_{cyc} \cdot (N_{miss,rd} \cdot N_{cyc,rd_pen} + N_{miss,wr} \cdot \\
& N_{cyc,wr_pen} + N_{miss,fet} \cdot N_{cyc,fet_pen}) \qquad (2)
\end{aligned}
$$

In the equation above T_{w/o_c} is the execution time of a program running on the processor core (simulated using an ISS) with a very large cache that would not cause any cache miss. However, the actual execution time (of a subsystem that has a real-world cache i.e. with limited cache size) is given by T_{w_c}. For more information on performance estimation please refer to [28].

5 Results

We have validated the I-CoPES methodology by means of a set of seven applications. The applications have been chosen with as much variety in characteristics as possible in order to show the wide application area of the methodology. Thus, the applications varied in size (8k to 200k), application area (video, animation, algorithmic etc.) and application domain (data dominated or control dominated). The applications used were: a complete MPEGII video encoder *mpeg*, the problem of locating eight queens on a chess board without interfering *q8* (we used 11 queens to make the problem reasonably big), a video trick animation algorithm *trick1*, the Whetston benchmark sequences *whetston*, the unix command compress *compress*, a chromakey video mixer as part of a digital video studio equipment and the travelling salesman problem *tsp*.

The results are given in Table 2. The first column gives the application, the second the cache size used for the application, the third the number of instructions for that application, the fourth, the number of instruction misses for that application with that particular cache for the FIFO case as compiled by "cc", the fifth the miss ratio for the FIFO case, the sixth the number of misses for the I-CoPES methodology, the seventh the miss ratio for the I-CoPES methodology, the eighth the energy consumed for the FIFO case, the ninth the energy consumed for the system under the I-CoPES methodology and finally the tenth column gives the energy reduction as a percentage.

As can be seen from the table, for most of the applications, there is a substantial reduction in miss ratios. For example, the miss ratio has reduced from 47.85% to 36.35% for the *mpeg* application with 512 bytes of cache. For the smaller cache sizes, the miss ratios are not reduced substantially since the application has several basic blocks which are comparable in size to the size of the cache. A more dramatic reduction in miss rates can be seen in the application called *trick1*, where for the case with cache size of 1024, the miss rates plunge from 71% to 0.04%. The miss rates for the *tsp* problem were reasonably high, and this was because a few long loops were executed repeatedly. Such a case requires a lot of swapping of the cache, and therefore the miss rates did not substantially change.

The total memory sizes increased by between 3 and 38% (not shown in the table). The average increase was 13%. However, in an embedded system which is on a single chip, the unused memory locations can be removed, and therefore there will be no real increase in memory size and consequently the chip size will remain almost the same. There is a possibility, that an additional line might be needed for the address bus, and therefore at the extreme case, we could expect a small increase in chip size.

The energy consumption reduction is also substantial, as can be seen from the last column. There are a few cases where the energy consumed has increased, but these are for cases where the miss rates are relatively small. The increase is due to the increased switching which is the result of the reordering of blocks in memory. The energy consumption in the cases where it has increased has never been more than 4.5% and where it has decreased, it has decreased by up to 65%. The maximum energy saving for each application is given in Figure 5.

Though we are focusing on the reduction of energy and obtain mostly significant reductions, this is NOT at the cost of performance. In fact, performance increases in all cases as the numbers

Table 2: Table for results

Application	I-Cache Size	# of Inst.	FIFO miss	Miss Ratio	I-CoPES Miss	I-CoPES Miss Ratio	FIFO (J)	I-CoPES Energy (J)	Energy Sav %
mpeg	128	22406459	16343210	72.94%	15985131	71.34%	23.966566	23.459693	2.11%
	256	22406459	12592796	56.20%	11486547	51.26%	18.91491	17.35585	8.24%
	512	22406459	10721491	47.85%	8143642	36.35%	16.424782	12.887134	21.54%
	1024	22406459	7378424	32.93%	4654360	20.77%	11.969864	8.1840637	31.63%
	2048	22406459	582241	2.60%	310957	1.39%	2.8494426	2.4016655	15.71%
Q8	64	44814000	30307447	67.63%	23340791	52.08%	4.7998051	3.8059608	20.71%
	128	44814000	27500971	61.37%	11381784	25.40%	4.4282823	2.1563028	51.31%
	256	44814000	4538800	10.13%	3772601	8.42%	1.21218817	1.11561781	7.97%
	512	44814000	241	0.00%	241	0.00%	0.62180677	0.63018477	-1.35%
trick1	128	103104786	103104786	100.00%	103104786	100.00%	19.514151	19.602431	-0.45%
	256	103104786	102669907	99.58%	100175851	97.16%	19.582832	19.368942	1.09%
	512	103104786	90421317	87.70%	64081456	62.15%	18.081579	14.2167598	21.37%
	1024	103104786	73378138	71.17%	39026	0.04%	16.152711	5.53378491	65.74%
	2048	103104786	18248122	17.70%	486	0.00%	8.673339	5.7163015	34.09%
whetston	128	1749402	1701363	97.25%	1635829	93.51%	0.29518329	0.28625952	3.02%
	256	1749402	1215194	69.46%	836405	47.81%	0.22842737	0.1776056	22.25%
	512	1749402	108059	6.18%	106669	6.10%	0.073333428	0.076303336	-4.05%
	1024	1749402	993	0.06%	993	0.06%	0.062205727	0.064868727	-4.28%
compress	128	53280973	39686614	74.49%	28533943	53.55%	7.494007	5.9797702	20.21%
	256	53280973	29161991	54.73%	10489863	19.69%	6.0605139	3.4465544	43.13%
	512	53280973	7102452	13.33%	1555604	2.92%	3.0002462	2.23145635	25.62%
	1024	53280973	1272890	2.39%	256458	0.48%	2.2959965	2.16208414	5.83%
	2048	53280973	443168	0.83%	53789	0.10%	2.4338514	2.391145	1.75%
key	64	9849864	8123533	82.47%	7672106	77.89%	1.9128616	1.8714207	2.17%
	128	9849864	4299216	43.65%	3221671	32.71%	1.3871321	1.26189137	9.03%
	256	9849864	1744685	17.71%	1407958	14.29%	1.03327393	1.01142947	2.11%
	512	9849864	779934	7.92%	416077	4.22%	0.90934229	0.88227793	2.98%
	1024	9849864	95099	0.97%	13254	0.13%	0.8351728	0.8299966	0.62%
	2048	9849864	52786	0.54%	3595	0.04%	0.87677332	0.87650856	0.03%
tsp	64	3403735	2657486	78.08%	2622434	77.05%	0.51087595	0.50016747	2.10%
	128	3403735	1931083	56.73%	1902922	55.91%	0.41021391	0.40003662	2.48%
	256	3403735	1240095	36.43%	1180148	34.67%	0.31553183	0.30108842	4.58%
	512	3403735	856395	25.16%	778015	22.86%	0.26643111	0.25965276	2.54%
	1024	3403735	488049	14.34%	495250	14.55%	0.22323306	0.22760796	-1.96%
	2048	3403735	288144	8.47%	362509	10.65%	0.212027719	0.22134353	-4.39%

of cache miss ratios show (please compare column "Miss Ratio" and "I-CoPES Miss Ratio").[1].

Please note that we used two different processor energy models for the applications. So, not all of the applications can be compared with each other in terms of average energy consumption per instruction. Within each application, for differing cache sizes, we used the same processor model. An example is the differing energy levels between *mpeg* and *q8*. If the same processor energy model would have been applied we should expect total energy to be larger for *q8* since it has far more instructions. However, the applications don't have to be compared with each other (in terms of absolute energy consumption) as they would usually run on different processors, since the applications stem from different domains with different constraints. That does not limit the evaluation results at all but is important to mention if one were to compares absolute numbers between applications.

These performance figures are comparable to the figures given in [25], for the direct mapped case (they also explored other set associative caches). While we cannot compare the results directly with theirs (due to differing applications), the rates of reductions in miss rates are in the same order. They used an integer linear programming methodology, which takes a very long time for most applications. Their work did not look at power/energy reductions. The runtime for all of these applications varied from 1 - 8 minutes on a Sparc 10 machine. A much more substantial time (in the order of hours) was spent on extracting the instructions which formed the basic blocks and the loops which were made of these basic blocks. up of these basic block.

6 Conclusions

We have presented a technique that places instruction codes wisely into the main memory such that the subsequent mapping to the instruction cache leads to reduced cache miss ratio and thus to a power/energy minimization of the whole system (CPU, caches main memory, buses). The technique is divided into two algorithms: first we indentify an efficient code placement for the instruction in the I-cache and through the known and fixed mapping from main memory to the I-cache we reverse-calculate the according placement of the code for the main memory. Both algorithms are performed upfront i.e. before the system is executing and thus they do not impose any computational overhead to the system. One of our assumptions is that the task set that is running on the system is known a priori what is valid for many embedded systems. We

[1] There is only one exception i.e. application *tsp* with an I-cache size of 2048 but this would not be an appropriate cache size for this application anyway

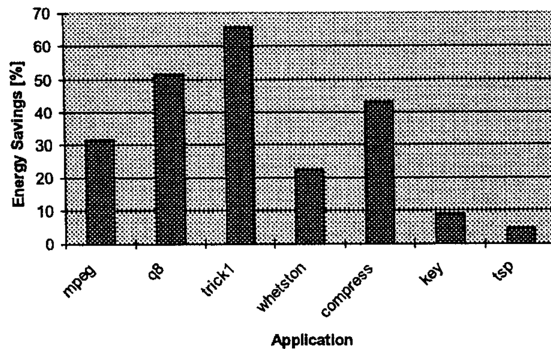

Figure 5: Final results: shown are the best energy savings for each application

have validated our techiques with seven applications that range in size from 8k to 200k. The technique leads to a reduction of energy/power consumption of up to 65% for a whole system. This is at the cost of an average main memory increase of 13%.

References

[1] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, *IMPACT: an architectural framework for multiple-instruction-issue processors* in Computer Architecture News. vol.19, no.3; May 1991;

[2] D. Liu and C. Svensson, *Impact of supply voltage on power consumption, speed, and reliability of CMOS circuits*, , 1994.

[3] S. McFarling, *Program optimization for instruction caches*, in ASPLOS III Proceedings. Third International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, New York, NY, USA; 1989; x+303 pp. p.183 91, 1989.

[4] S. McFarling, *Procedure merging with instruction caches*, in SIGPLAN Notices. vol.26, no.6; June 1991; p.71

[5] P. Panda, N. Dutt, and A. Nicolau, *Memory organization for improved data cache performance in embedded processors*, , 1996.

[6] P. R. Panda, N. D. Dutt, and A. Nicolau, *Efficient utilization of scratch-pad memory in embedded processor applications*, , 1997.

[7] P. R. Panda and N. D. Dutt, *Behavioral array mapping into multiport memories targeting low power*, , 1997.

[8] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, *Improving cache performance through tiling and data alignment*, , 1997.

[9] P. R. Panda, N. D. Dutt, and A. Nicolau, *Memory data organization for improved cache performance in embedded processor applications*, ACM Transactions on Design Automation of Electronic Systems, vol. 2, pp. 384-409, 1997.

[10] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, *Augmenting loop tiling with data alignment for improved cache performance*, IEEE Transactions on Computers, vol. 48, pp. 142-149, 1999.

[11] P.-P. Ranjan, H. Nakamura, N. D. Dutt, and A. Nicolau, *A data alignment technique for improving cache performance*, , 1997.

[12] L. Yanbing and W. Wolf, *Hardware/software co-synthesis with memory hierarchies*, Design Automation Conference, 1998.

[13] L. Yanbing and W. H. Wolf, *Hardware/software co-synthesis with memory hierarchies*, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 18, pp. 1405-1417, 1999.

[14] B.P. Dave, G. Lakshminarayana, N.K. Jha, *COSYN: Hardware-Software Co-Synthesis of Embedded Systems*' Proc. DAC'97, pp.703-708, 1997.

[15] I. Hong, D. Kirovski et al., *Power Optimization of Variable Voltage Core-Based Systems*, IEEE Proc. of 35th. Design Automation Conference (DAC98), pp.176-181, 1998.

[16] T. Givargis, F. Vahid, J. Henkel, *A Hybrid Approach for Core-Based System-Level Power Modeling*, Proc. ASP-DAC'99, pp.141-145, 1999.

[17] T. Simunic, L. Benini, G. De Micheli, *Cycle-Accurate Simulation of Energy Consumption in Embedded Systems*, Proc. DAC'99, pp.867-872, 1999.

[18] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, *Efficient Power Co-Estimation Techniques for System-on-Chip Design*, Proc. DATE'00, pp.27-34, 2000.

[19] M. D. Hill, J. R. Laurus, A. R. Lebeck et al., *WARTS: Wisconsin Architectural Research Tool Set*, Computer Science Department University of Wisconsin.

[20] F. Chow, *A portable machine-independent global optimizer—Design and measurements*, Tech. report 83-254, PhD thesis, Computer Systems Lab, Stanford Univ., 1983.

[21] D. B. Kirk, *SMART (strategic memory allocation for real-time) cache design*, in Proceedings. Real Time Systems Symposium (Cat. No.89CH2803 5). IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1989; pp. p.229 37, 1989.

[22] D. B. Kirk and J. K. Strosnider, *SMART (strategic memory allocation for real-time) cache design using the MIPS R3000*, in Proceedings. 11th Real Time Systems Symposium (Cat. No.90CH2933 0). IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1990; xi+341 pp. p.322 30, 1990.

[23] D. Kirovski, L. Chunho, M. Potkonjak, and S.-W. H. Mangione, *Application-driven synthesis of memory-intensive systems-on-chip*, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 18, pp. 1316-1326, 1999.

[24] Y. T. Li, S. Malik, and A. Wolfe, *Performance estimation of embedded software with instruction cache modeling*, ACM Transactions on Design Automation of Electronic Systems, vol. 4, pp. 257-279, 1999.

[25] H. Tomiyama and H. Yasuura, *Code placement techniques for cache miss rate reduction*, ACM Transactions on Design Automation of Electronic Systems, vol. 2, 1997.

[26] C. Kulkarni, F. Catthoor, and H. De Man, *Code transformations for low power caching in embedded multimedia processors*, Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing IEEE Comput. Soc, Los Alamitos, CA, USA; p.292-7 , 1998.

[27] F. Catthoor, N. D. Dutt, and C. E. Kozyrakis, *How to solve the current memory access and data transfer bottlenecks: at the processor architecture or at the compiler level?*,Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 , 2000.

[28] Y. Li and J. Henkel, *A framework for estimating and minimizing energy dissipation of Embedded HW/SW Systems*, IEEE/ACM 35th. Design Automation Conference (DAC'98) 1998, pp.188-193, 1998.

[29] S. Parameswaran, *Code placement in Hardware Software Co-Synthesis to Improve Performance and Reduce Cost*, Design Automation and Test in Europe, pp. 627-633, 2001.